JavaScript Promises

1. What *is* a Promise?

A Promise is JavaScript's built-in "IOU". It represents the *future* result of an asynchronous operation.

Analogy → Job interview

1. You finish the interview (kick off an async task).
2. The interviewer says, "We'll let you know within a week" (returns a Promise).
3. Later you either get:

• an **offer** (fulfilled)
• a **rejection** (rejected)
• or you're still waiting (pending)

2. The three states

State	Description	Typical moment in code
Pending	Work in progress; no result yet	const p = fetch(url) immediately after call
Fulfilled	Operation succeeded → value is available	The resolve() branch of a Promise executor
Rejected	Operation failed → reason (error) is available	The reject() branch or a thrown error

3. Creating Promises from scratch

```
const myPromise = new Promise((resolve, reject) => {
  // async work
  if (success) resolve(data);      // moves to "fulfilled"
  else reject(new Error("oops"));  // moves to "rejected"
});
```

Most modern APIs (*fetch*, *Web Crypto*, *File System Access*, etc.) create the Promise for you, so you rarely need `new Promise` yourself.

4. Consuming Promises

4.1 `.then()`

```
promise.then(onFulfilled[, onRejected]);
```

- returns a **new** Promise
- if you return a value → it becomes the next `.then()`’s input
- if you return another Promise → the chain *waits* for it

```
fetch("https://apis.scrimba.com/bored/api/activity")
  .then(res => res.json())     // ← returns a Promise
  .then(data => console.log(data.activity));
```

4.2 Value *propagation* example (your “Hello → World” chain)

```
fetch("https://apis.scrimba.com/bored/api/activity")
  .then(() => "Hello")          // resolved value is now "Hello"
  .then(word => {
      console.log(word);        // → Hello
      return "World";
  })
  .then(next => console.log(next)); // → World
```

Each `.then()` gets whatever the previous one *returned* (or *threw*).

4.3 `.catch()` — error branch

```
fetch(url)
   .then(handleOK)
   .catch(err => console.error("Something went wrong", err));
```

`.catch(fn)` is equivalent to `.then(null, fn)` and it also returns a Promise, so you can keep chaining after a catch.

4.4 `.finally()` — cleanup

Runs regardless of outcome.

```
doSomething()
  .finally(() => spinner.hide());
```

5. Method chaining outside Promises

You already did:

```
document.getElementById("new-deck")
        .addEventListener("click", () => console.log("Clicked"));
```

Same spirit: each call returns an object you can immediately call more methods on.

6. Array *data* chaining reminder

```
const emails = voters                    // start array
  .filter(p => p.voted)                  // keep true voters
  .map(p => p.email);                    // pluck email
```

`.filter()` returns a new array → `.map()` can run right away.

7. Aggregate Promise helpers

Helper	Resolves with...	Rejects when...
`Promise.all([...])`	array of *all* results (order preserved)	*any* input rejects
`Promise.race([...])`	first settled result (value or error)	first result is rejected & no earlier fulfill

`Promise.allSettled([...])`	array of objects `{status, value	reason}`
`Promise.any([...])`	first fulfilled value	rejects if *every* input rejects

8. async/await — syntactic sugar over Promises

```
async function getActivity() {
  try {
    const res = await
fetch("https://apis.scrimba.com/bored/api/activity");
    const data = await res.json();
    console.log(data.activity);
  } catch (err) {
    console.error(err);
  }
}
```

Rules:

1. `await` only valid inside `async` functions.
2. `await` pauses the function until the Promise settles, **without** blocking the main thread.
3. An async function always returns a Promise (its fulfilled value is whatever you `return`).

9. Sequential vs parallel

```
// sequential
await task1();
await task2();      // starts *after* task1

// parallel
const [a, b] = await Promise.all([task1(), task2()]);
```

Understanding this difference can save seconds (or minutes) in network-heavy apps.

10. Common pitfalls

• Forgetting to `return` inside `.then()` → chain receives `undefined`.

• Throwing inside an executor without `try/catch` → unhandled rejection.

• Nested callbacks inside `.then()` instead of returning a Promise → “Promise hell”.

• Using `await` on non-Promise values (harmless but redundant).

11. Promisifying callback APIs

```
function readFileAsync(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, "utf8", (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
}
```

Node’s `fs.promises` already did this, but the pattern is handy for older libraries.

12. The microtask queue (event loop refresher)

• When a Promise settles, its `.then/.catch/.finally` callbacks enqueue as **microtasks**.

• Microtasks run **right after** the current JavaScript call stack finishes, *before* the browser processes rendering or `setTimeout` callbacks.

13. Cheat-sheet recap

```
States: pending → fulfilled OR rejected
.then(success?, fail?)      // returns a new Promise
.catch(fail)                // sugar for .then(null, fail)
.finally(cleanup)           // runs in either case
Promise.all / race / allSettled / any
async / await               // write sync-style async code
```

14. Best practices

1. Always end promise chains with `.catch()` (or wrap in `try/await/catch`).
2. Return results from `.then()` instead of nesting.
3. Prefer `async/await` for readability; fall back to raw Promises for fine-grained control.
4. Use `Promise.all` for truly parallel tasks; otherwise loop with `for…of` + `await` to stay sequential.
5. Handle *both* network and parsing errors (`res.ok` check + `try/await/catch`).